Schema-based Query Optimization for XQuery Queries

Sven Groppe and Stefan Böttcher

University of Paderborn, Faculty 5, Fürstenallee 11, D-33102 Paderborn, Germany
{sg, stb}@uni-paderborn.de

Abstract. XQuery is widely used for querying XML documents. Within this paper, we examine optimization rules for XQuery queries that exploit type information of the input XML document given in XML Schema. These optimization rules are applicable for all XQuery expressions and are very useful e.g. in the scenario of XQuery queries on XQuery views. The basic idea is to transform the XML Schema definition into a graph, which is extended to a graph representing the XQuery expression. The latter graph is used to delete subexpressions of the XQuery expression that are not used to retrieve the final result of the given XQuery expression. We further include experimental results that demonstrate the improvement of our optimization.

1 Introduction

The W3C has developed XQuery [25] for querying XML documents in recent years. XQuery evaluators have not only been implemented for single XML documents, but XQuery has also been integrated in many XML databases and XML-enabled databases. However, query optimization for XQuery is still a major challenge.

In this paper, we introduce a new optimization technique for the following class of XQuery expressions. We call the class of XQuery expressions C_{XQuery}, which consist of all XQuery expressions, which contain at least one sub-expression, which generates either no output at all or an intermediate result, which is not used to compute the final result of the query. Our optimization technique identifies and eliminates those sub-expressions of queries of C_{XQuery} which do not generate any output or generate superfluous intermediate results.

Even if queries are well designed, queries of C_{XQuery} often occur in the following important scenario of queries on views. A view describes how a certain section of the stored data in the database is transformed. Now, a user can query the view to retrieve all or a subset of the transformed data of the view. Inside database management systems, the query on the view and the view itself are composed and then evaluated as one composed query. Whenever the user restricts the view by the query and does not query for all the data of the view, the composed query can in general belong to the class C_{XQuery}. In these cases, applying our new optimization technique saves time of processing the composed query. We show by experimental results that the execution of the optimized query is much faster than computing the complete query, whenever the size of all data is relatively large and we can avoid unnecessary computation of unneeded intermediate results.

The rest of the paper is organized as follows. Section 2 outlines an optimization example. Section 3 describes our general approach for optimizing XQuery queries. Section 4 presents a performance analysis. Section 5 refers to the related work, and we end up with the summary and conclusions in Section 6.

2 Optimization Example

In order to illustrate our optimization, we start with an example. Let us consider the following XQuery query in Fig. 1, which is composed of a view definition in line (1) to line (19) and of a user defined query on this view in line (21). The XQuery query in Fig. 1 shall be applied to the input XML document of Fig. 2 that fulfills the XML schema definition of Fig. 3.

```
(1)let $view :=
(2) <root>
(3)  <result>
(4)  {
(5)   for $a in
(6)       document('p.xml')/conference/paper
(7)   let $b :=
(8)   <single_result>{$a}</single_result>
(9)   return $b
(10) }
(11) {
(12)  for $a in
(13)      document('p.xml')/conference/tutorial
(14)  let $b :=
(15)  <single_result>{$a}</single_result>
(16)  return $b
(17) }
(18) </result>
(19)</root>
(20)return
(21) $view/result/single_result/tutorial
```

Fig. 1. Query for retrieving the tutorials

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE conference SYSTEM "DTD\proceedings.dtd">
<conference name="All Topics">
  <tutorial name="Web Services">
    <author>Expert</author>
  </tutorial>
  <paper
   name="Solving unsolved problems">
    <author>Problem Solver</author>
    <author>Problem Searcher</author>
  </paper>
</conference>
```

Fig. 2. Input XML document `p.xml`

```
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
<xsd:element name='conference'>
```

```
 <xsd:complexType>
  <xsd:choice minOccurs=1 maxOccurs='unbounded'>
    <xsd:element name='tutorial'>
     <xsd:complexType>
      <xsd:element ref='author' minOccurs=0 maxOccurs='unbounded'/>
     </xsd:complexType>
     <xsd:attributeGroup ref='name'/>
    </xsd:element>
    <xsd:element name='paper'>
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element ref='author' minOccurs=0 maxOccurs="unbounded"/>
       <xsd:element name='references' minOccurs=0 maxOccurs=1>
        <xsd:complexType>
         <xsd:element ref='conference' minOccurs=0
          maxOccurs='unbounded'/>
        </xsd:complexType>
       </xsd:element>
      </xsd:sequence>
     </xsd:complexType>
     <xsd:attributeGroup ref='name'/>
    </xsd:element>
  </xsd:choice>
 </xsd:complexType>
 <xsd:attributeGroup ref='name'/>
</xsd:element>
<xsd:element name='author'>
 <complexType mixed='true'/>
</xsd:element>
<xsd:attributeGroup name='name'>
 <xsd:attribute name='name' use='required'/>
</xsd:attributeGroup>
</xsd:schema>
```

Fig. 3. Schema of input XML document `p.xml`

When the XQuery query in Fig. 1 is evaluated on the XML document '`p.xml`' outlined in Fig. 2 and its XML Schema definition outlined in Fig. 3, at first the variable `$view` is computed. Its content is presented in Fig. 4.

```
<root>
 <result>
  <single_result>
   <tutorial name="Web Services">
     <author>Expert</author>
   </tutorial>
  </single_result>
  <single_result>
   <paper
    name="Solving unsolved problems">
     <author>Problem Solver</author>
     <author>Problem Searcher</author>
   </paper>
  </single_result>
 </result>
</root>
```

Fig. 4. Content of `$view` when evaluating the query of Fig. 1

Thereafter, the result of the entire query of Fig. 1 is computed, the result of which is presented in Fig. 5.

```
<tutorial name="Web Services">
  <author>Expert</author>
</tutorial>
```

Fig. 5. Result of the query in Fig. 1

Note that the result in Fig. 5 is only a part of the fragment computed by the variable `$view`. Although XQuery evaluators process the complete query in Fig. 1, and therefore can optimize the complete query, current implementations ([6, 15, 18, 22]) always compute the *entire* content of the variable `$view` either at once or one XML node after the next XML node by using an iterator. After that, current implementations project to the required part of `$view`.

In comparison, our approach computes a variable assignment of `$view` which only contains those computations for the *required* part, which is used to compute the final result. For example, the query in Fig. 6 is an optimized query of the query in Fig. 1. Within Section 4, we present experimental results, which show the speed-up factor of such optimized queries.

```
let $view :=
<root>
 <result>
 {
  for $a in
   document('p.xml')/conference/tutorial
  let $b :=
   <single_result>{$a}</single_result>
  return $b
 }
 </result>
</root>
return $view/result/single_result/tutorial
```

Fig. 6. Optimized query of the query in Fig. 1

In the following subsections, we describe how to eliminate sub-expressions in variable assignments like `$view` so that only the required part is computed. This optimization technique can be used in the important scenario of XQuery queries on XQuery views, because one way to reformulate an XQuery query `Q` according to an XQuery view `V` is to use the pattern presented in Fig. 7. Whenever there is a reference in the XQuery query `Q` to the view `V`, the variable `$view` is used to access the view.

```
let $view := <root> { V } </root>
return Q
```

Fig. 7. Pattern of reformulating an XQuery query `Q` according to an XQuery view `V`

3 The General Optimization Approach

We design the optimization step of reducing the XQuery query independently of the current instance of the XML document. The advantage is that the approach can optimize the XQuery query in advance without connecting to any database. As the output of XQuery expressions often contains whole sub-trees of the input XML document,

we can use schema information of the input XML document in order to optimize queries. For this purpose, we introduce the ordered schema graph, which is based on the schema of the input XML document. We use this ordered schema graph for search algorithms, which are outlined in Section 3.2 and Section 3.4.

3.1 Ordered Schema Graph

The ordered schema graph is generated from an XML Schema definition. As an example, see the XML Schema definition in Fig. 3 and the ordered schema graph in Fig. 8.

Each node of an ordered schema graph represents an element node of the schema, the document node or a dummy node (called `:EMPTY` node). The `:EMPTY` node represents a whole choice expression or a whole sequence expression. There are three kinds of edges: *parent-child edges* represent a parent child relationship between element nodes, the document node and/or `:EMPTY` nodes. A *sibling edge* represents a directed sibling relationship between element nodes, the document node and/or `:EMPTY` nodes. Finally, an *expression edge* represents a relationship to a whole choice expression or a whole sequence expression between element nodes, the document node and `:EMPTY` nodes.

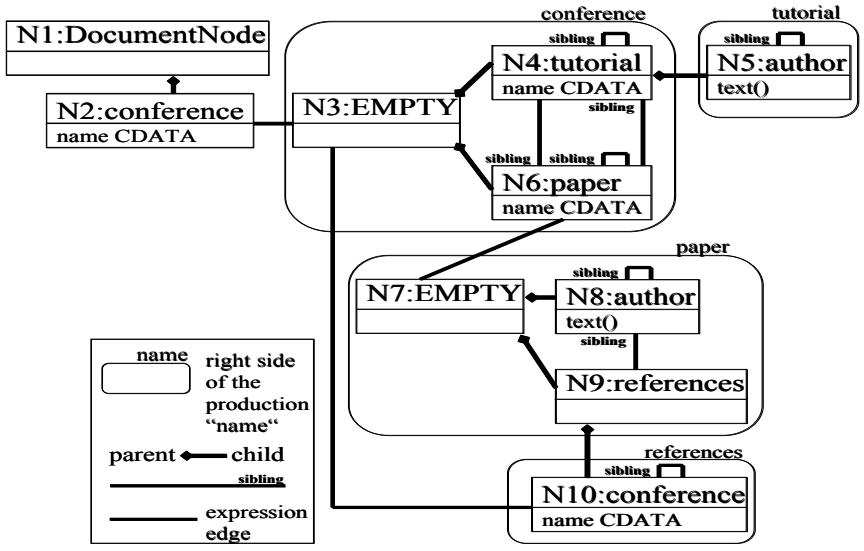

Fig. 8. Ordered schema graph of the schema defined in Fig. 3

We present here the general rules for generating the ordered schema graph from an XML Schema definition:

- We create a start node of type `:DocumentNode`, the child of which is a node representing that element, which is the root node of the XML document. In the example of Fig. 3 and Fig. 8, `N1` represents the document node and `N2` represents the root element node.

- We add all (required, implied and fixed) attributes of any type of the corresponding XML element `E` to the nodes representing `E`. In the example of Fig. 3 and Fig. 8, we add the attribute `name` to the node `N2`.
- We transform the right-hand side of an element declaration according to the following rules:
    - Nodes of the ordered schema graph representing elements are the parents of the representation of the right-hand sides of their element declarations. In the example of Fig. 3 and Fig. 8, `N1` is the parent of `N2`.
    - Whenever an element `E1` can be a following sibling node of another element `E2`, we insert a sibling edge from `E2` to `E1`. This is the case for repetitions of elements (see right-hand sides of element declarations of `conference`, `tutorial`, `paper` and `references` in Fig. 3 and Fig. 8) and for sequences of elements (see right-hand side of `paper`).

Whenever the XML Schema defines an element to be a `complexType` defined by a choice (see right-hand side of `conference`) or a sequence (see right-hand side of `paper`), then we create an extra `:EMPTY` node for easy access to the whole choice expression or the whole sequence expression, respectively. As an example, see node `N3` in Fig. 8 representing the `xsd:choice` element in Fig. 3 and node `N7` representing the `xsd:sequence` element in Fig. 3.

3.2 Satisfiability of an XPath Expression According to a Schema

Definition: An XPath expression `XP` is *satisfiable* according to a schema, if and only if there exists at least one document, which is valid according to the schema, where `XP` is evaluated to a non-empty result.

The problem of satisfiability of certain subclasses of XPath expressions (without respect to a schema) is in NP [14]. We present here a fast (but incomplete) satisfiability test for XPath expressions according to a schema. The test is incomplete in the following way. The test returns *not satisfiable* if we are sure that the XPath expression is not satisfiable according to a schema. Otherwise the test returns *may be satisfiable*.

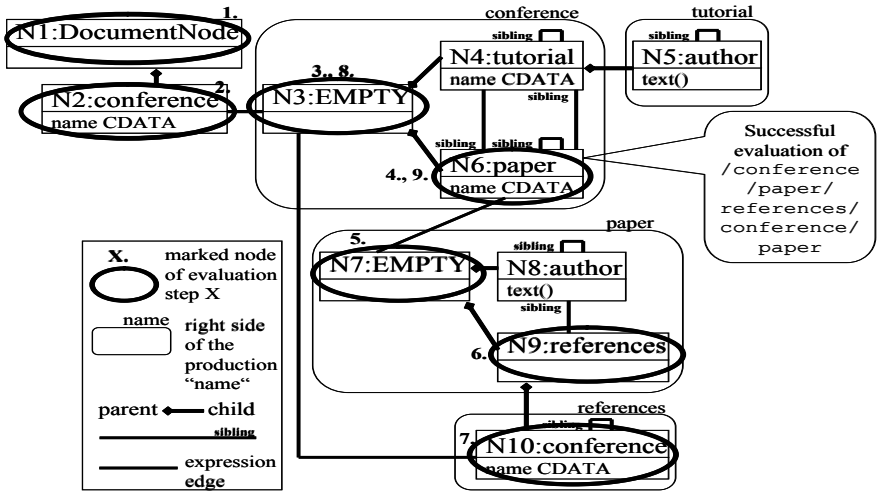

Fig. 9. Ordered schema graph with marked nodes according to the query `conference/paper/references/conference/paper`

We use a modified XPath evaluator to test whether or not the XPath expression `XP` is satisfiable according to a schema. The input of the modified XPath evaluator is the schema and `XP`. First, the ordered schema graph is created according to the schema by using the rules summarized in Section 3.1. After that, we execute the modified XPath evaluator, which performs the task to check whether or not all elements and attributes of paths within `XP` are available at the required hierarchical position. The modified XPath evaluator starts at the node of type `:DocumentNode` representing the document node. Within the ordered schema graph, there is all necessary information in order to execute the modified XPath evaluator, as the parent-child-axis and the next-sibling-axis are available in the ordered schema graph. The evaluator passes empty nodes without consuming an element of the XPath expression `XP`. In comparison to an XML document, the ordered schema graph can contain loops. Therefore, the modified XPath evaluator must consider loops when it revisits a node but did not process the next location step within `XP`. For this purpose, the modified XPath evaluator marks all the nodes that contribute to a successful evaluation of `XP`.

As an example, see the steps 1, …, 9 performed on the ordered schema graph of Fig. 9 for the successful evaluation of the XPath query `/conference/paper/references/conference/paper`. As in step 9 the whole XPath expression is processed, this XPath query is considered to be satisfiable.

In general, with this technique, we can test whether or not a schema definition allows only XML documents for which a given XPath expression `XP` can never be successfully evaluated, i.e. for which the evaluation of the XPath expression returns an empty set.

3.3 XQuery Graph

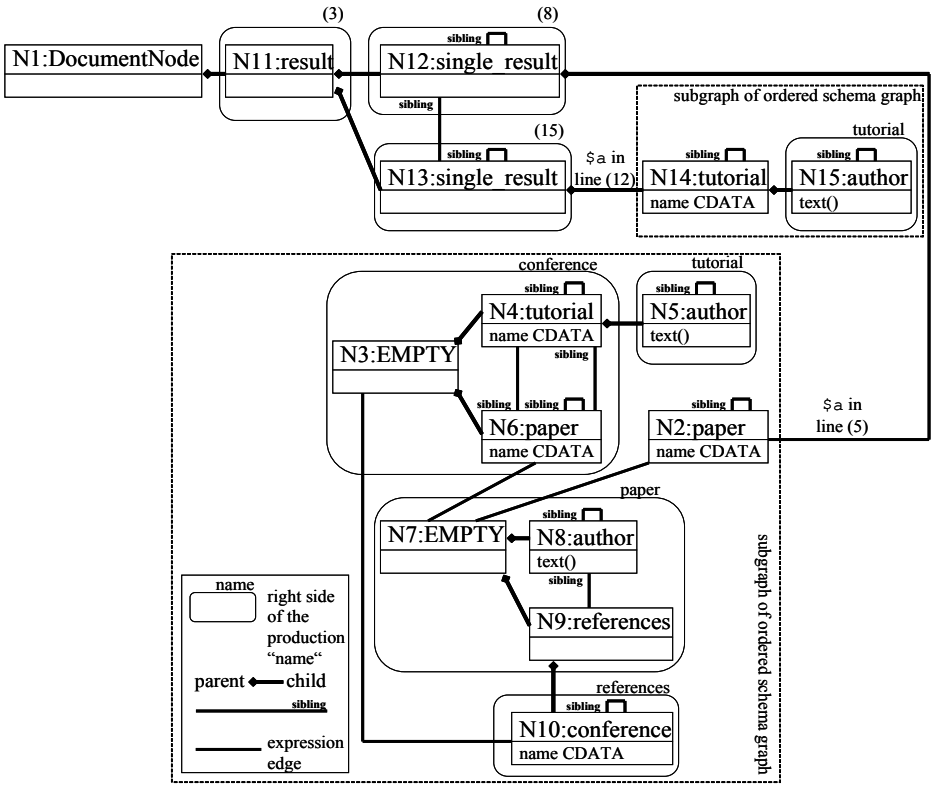

Fig. 10. XQuery graph of the XQuery query of Fig. 1

For the optimization rules, we consider that subset of XQuery, where the XQuery expression must conform to following rule `Start` in EBNF notation.

```
Start ::= (FunctionDecl)* FLRExpr.
FunctionDecl ::= "declare" "function" QName "(" ("$"
                 QName ("," "$" QName)*)? ")" "{" ExprSingle "}".
FLRExpr ::= (ForClause | LetClause)+ "return" ExprSingle.
ForClause ::= "for" "$" VarName "in" ExprSingle.
LetClause ::= "let" "$" VarName ":=" ExprSingle.
ExprSingle ::= FLRExpr|IfExpr|PathExpr.
IfExpr ::= "if" "(" ExprSingle ")" "then" ExprSingle "else" ExprSingle.
PathExpr ::= ("/" RelativePathExpr?) |
             ("//"RelativePathExpr)  | RelativePathExpr.
RelativePathExpr ::= (Step | PrimaryExpr)(("/"|"//")
                     (Step | PrimaryExpr))*.
Step ::= ("child" | "descendant" | "attribute" | "self" |
          "descendant-or-self" | "following-sibling" | "following" |
          "parent" |"ancestor" | "preceding-sibling" | "preceding" |
          "ancestor-or-self") "::" (QName | "node()" | "*").
```

```
PrimaryExpr ::= "$" QName | Constructor | FunctionCall.
Constructor ::= ("element" | "attribute") QName "{" ExprSingle "}".
FunctionCall ::= QName "(" (ExprSingle ("," ExprSingle)*)? ")".
```

This subset of XQuery contains nested `for-let-return` clauses, `if` expressions, element and attribute constructors, declarations of functions and function calls.

We present here the general rules for generating the XQuery graph from an XQuery expression: We generate an own XQuery graph for each variable assignment `$view` of the XQuery expression. Every expression within the variable assignment of `$view`, which generates output, gets its own new node `N` representing the output. Variables (and also nested variables) are replaced with their content. We set a new node `N` as parent node of every node in the XQuery graph representing output, which could be generated as child node to the output of node `N` by the XQuery evaluator. Furthermore, we relate the new node `N` with every node in the XQuery graph representing output, which could be generated as sibling node to the output of node `N` by the XQuery evaluator, by a directed sibling relation. If the next generated output of the XQuery evaluator is a sub-tree of the input XML document specified by an XPath expression `XP`, we search within the ordered schema graph by the modified XPath evaluator as before, and we retrieve a node set `SN` of nodes of the ordered schema graph. We first copy the nodes of `SN` and copy all descendant nodes and all sibling nodes, which can be reached from the nodes of `SN` by parent-child relationships and sibling relationships. We copy also all parent-child relationships and sibling relationships for the copied nodes. Finally, we set the current node as parent node of the copies of `SN`. In the example of Fig. 10, which represents the XQuery graph of the XQuery query of Fig. 1, these copies consists of the node `N2` and its descendant nodes `N3` to `N10`, and the node `N14` and its descendant node `N15`. We label the association with the XPath expression of the XQuery expression (or we use a reference; here, line number (5) and (12)).

3.4 Optimization

This section outlines how the XQuery graph is used to optimize queries. In the following paragraphs, we describe the general rules for optimizing the query.

The first step of the optimization approach is as follows: Whenever the content of a variable `$view` is queried by an XPath expression `XP` by `$view/XP` in the XQuery query, we process the following optimization steps. We execute the modified XPath evaluator on the XQuery graph of the variable assignment of `$view` with the input XPath query `XP`. In the case of the XQuery expression in Fig. 1, the XPath query `XP` is `/result/single_result/tutorial` for `$view`. The modified XPath evaluator marks all nodes within the XQuery graph that contribute to a successful evaluation of the query `XP` in the same way as an XPath query is evaluated on the ordered schema graph.

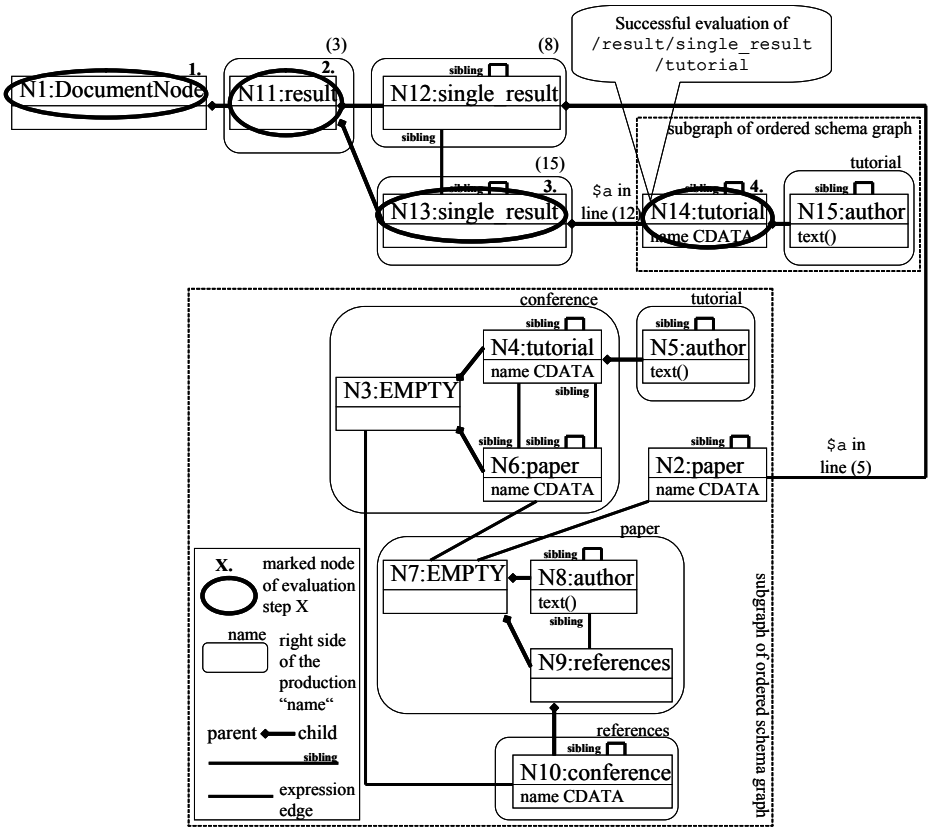

Fig. 11. Marked nodes of ordered schema graph for `/result/single_result/tutorial`

Within a second step, we do not delete all the sub-expressions within the XQuery expression from which the marked nodes in the XQuery graph are generated and which do not assign variables, which are used in sub-expressions of marked nodes. The `for`-statement defines how often and in which order the `result`-statement is executed. Therefore, we do not delete `for`-statements, except if their `return`-statement is reduced to an empty statement.

We delete all other unmarked sub-expressions and finally the optimized query remains.

The pseudo code of the entire algorithm for optimizing XQuery queries is given in Fig. 12, which expects an XQuery query `Q` as input and which returns the optimized XQuery query `Q'` as output.

In the case of the XQuery query in Fig. 1, we first generate the XQuery graph shown in Fig. 10 for the variable assignment of `$view`. Then we mark the nodes shown in Fig. 11, and finally we optimize to the XQuery query in Fig. 6.

```
Algorithm OptimizeQuery
Input: XQuery query Q conforming to rule Start in Section 3.3
Output:Optimized XQuery query Q'

(1)  Generate abstract syntax tree T of Q
(2)  Compute XQuery graph XG with marked nodes of T
(3)  Mark all nodes in T, which correspond to marked nodes in XG
(4)  while(all children of a symbol ExprSingle of a LetClause
     expression are unmarked) do
(5)          delete the whole LetClause expression
(6)  For all nodes n in T do
(7)          If(n and its child nodes are unmarked and
                  (n is a symbol ExprSingle and
                   not(n is a parameter of a function call or
                       n is a condition of an if-statement))
             ) then
(8)               delete n (and its children)
(9)  Compute Q' from the remaining nodes of T
```

Fig. 12. Algorithm for optimizing XQuery queries

4 Performance Analysis

The test system for all experiments is a 1.7 GHz Intel Pentium 4 processor with 128 Megabyte RAM, Windows 2000 as the operating system and Java VM build version 1.4.2. We use the XQuery evaluator of Saxon version 7.9 [15].

We have generated test input XML documents of different size valid according to the schema of Fig. 3 for all experiments, where every `paper` and `tutorial` element contains exactly one empty `author` element. Furthermore, we have used the XQuery query of Fig. 1 and the optimized query of Fig. 6. In the figures, we present the average of 10 experiments.

In the first experiment, the test input XML documents consist of the same amount of `paper` and `tutorial` elements. We increase the file size of the input XML documents from approximately 8 Kilobytes to approximately 8 Megabytes. Fig. 13 shows the evaluation time depending on the filesize in Kilobytes. The evaluation of the optimized query is approximately 1.8 times faster than the evaluation of the original query for file sizes larger than 1.5 Megabytes. Fig. 14 presents how much faster the evaluation of the optimized query is than the evaluation of the non-optimized query.

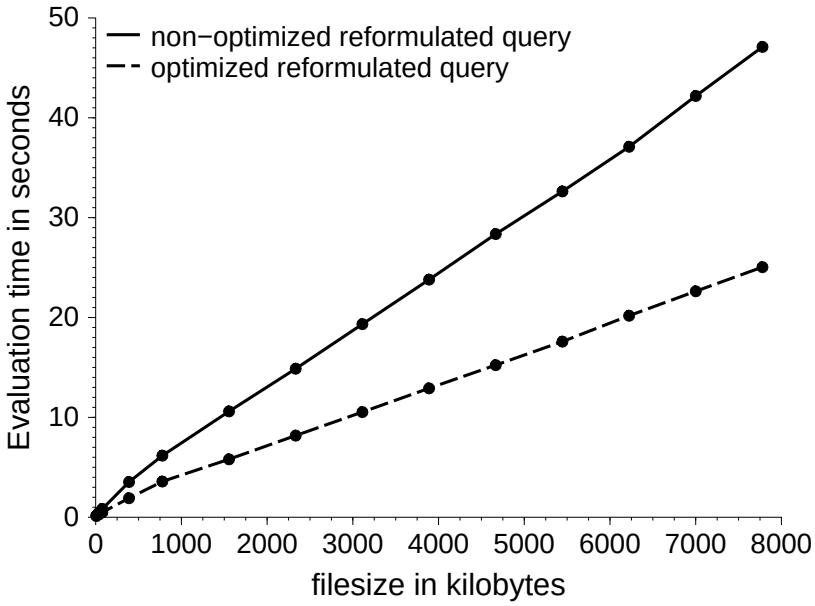

Fig. 13. Experiment 1: Evaluation time depending on filesize

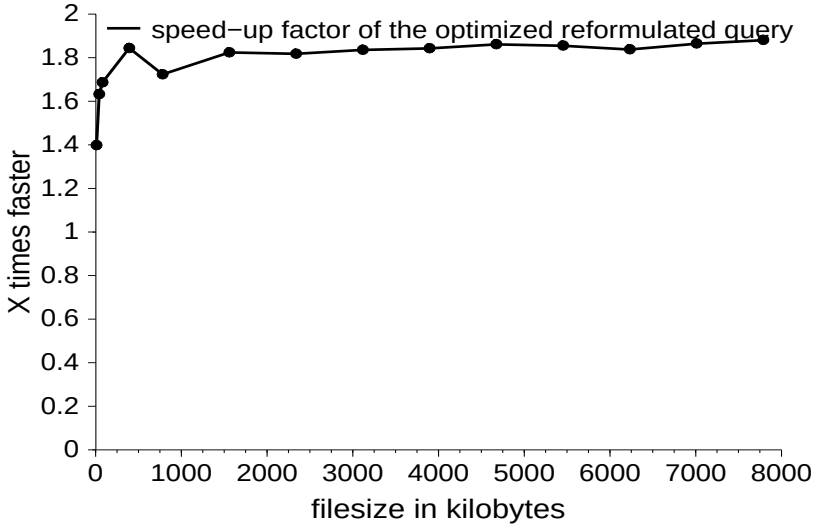

Fig. 14. Speed-up factor in Experiment 1

In the second experiment, the size of the input XML document is constant (here approximately 2 Megabytes), but we vary the amount of `paper` and `tutorial` elements (see Fig. 15). The speed-up factor of the optimized query varies from 14.6,

when there are no `tutorial` elements, to 1, when there are only `tutorial` elements (see Fig. 16).

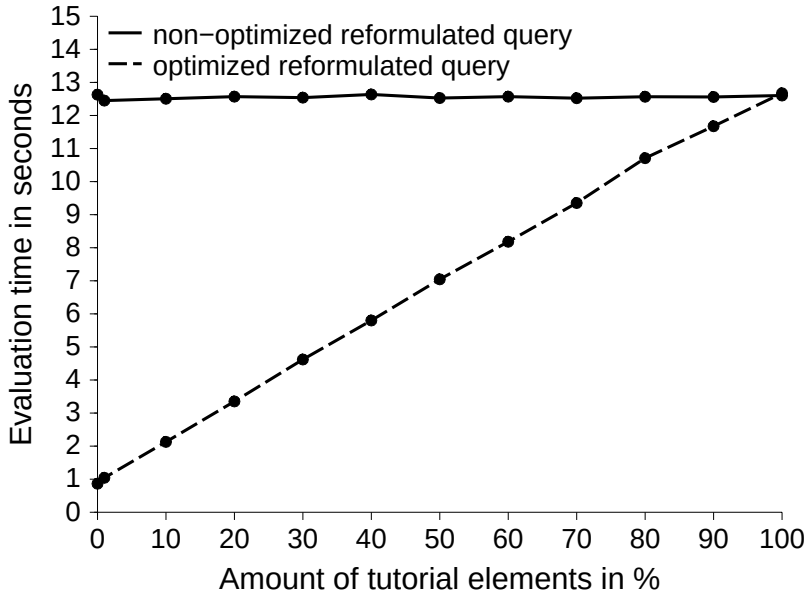

Fig. 15. Experiment 2: Evaluation time depending on the selectivity of the query for a constant filesize of 2 Megabytes

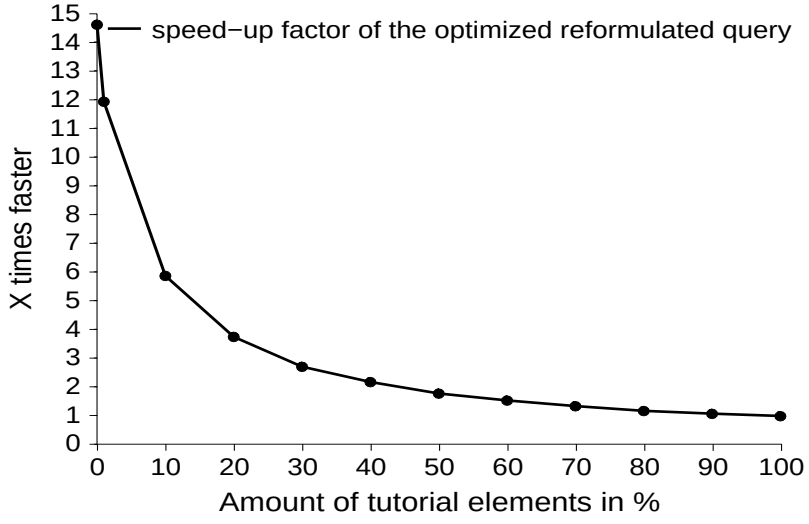

Fig. 16. Speed-up factor of Experiment 2

5 Related Work

The contributions [8, 10, 20] introduce an algebra for XQuery. Additionally, they list transformation and optimization rules based on the introduced algebra, but they do not contain the optimization approach presented in this paper.

[2] describes how the language XQuery can be extended to support views. It describes the language extensions but does not describe how to optimize.

[17] projects XML documents to a sufficient XML fragment before processing XQuery queries. It contains a static path analysis of XQuery queries, which computes a set of projection paths formulated in XPath. Our approach optimizes the XQuery expression itself and does not project XML documents.

Whereas the complexity of XPath query evaluation on XML documents is examined in [9], we consider the complexity of our XPath query evaluation algorithm on an XQuery graph, which is a part of the proposed optimization steps for XQuery queries.

[7] uses graph schemas to optimize regular path expressions within queries for semistructured data. In comparison, we do not optimize a path expression according to a schema, but avoid unnecessary transformation steps by eliminating query code for the generation of output, which is not used further. Furthermore, we introduce an ordered schema graph (and an XQuery graph as extended version for XQuery expressions), which contain additional information in comparison to graph schemas like a sibling relationship, and we deal with XPath as path language.

[14] deals with the problem of satisfiability of XPath expressions without respect to schema information as e.g. by an XML schema definition. In comparison, we introduce a fast (but incomplete) test that checks whether or not a given XPath expression is valid according to a given schema and according to a given XQuery expression.

[16] deals with the test of satisfiability of tree pattern queries, which cover a fragment of XPath, without respect to a schema and with respect to an acyclic schema. [16] discusses, when the test of satisfiability is NP-complete and when there exist polynomial time algorithm for the test of satisfiability. In comparison, we introduce a fast (but incomplete) satisfiability test according to a given schema, which can be also a cyclic schema, and according to a given XQuery expression.

Papakonstantinou et al. [19] studies the inference of DTDs for views of XML data, but uses the language loto-ql for the definition of XML views, which is less powerful than XQuery. Furthermore, our approach optimizes all XQuery queries and cannot be only used in the scenario of XQuery queries on XQuery views.

[4] investigates XML document specifications with schemas and integrity constraints. It deals with the consistency problem, i.e. whether or not there exists an XML document that both conforms to the schema and satisfies the constraints. In comparison, we investigate XPath and XQuery expressions and present a fast (but incomplete) test that checks whether or not a given XPath expression is valid according to a given schema.

[5, 21] describe frameworks for publishing relational data in XML. Both frameworks allow specifying view definitions formulated in XQuery, and optimize the reformulated query of a user-defined query according to a view by deleting unneces-

sary parts. We follow this idea; however, we optimize even more, as we also consider the restrictions of copied sub-trees of the input XML document within the user-defined query and view.

Our work was inspired by contributions in [11, 12, 13], which deal with XPath query reformulation according to an XSLT view and its optimization. In comparison, in this paper we describe general optimization rules, which are especially applicable for query reformulation on an XQuery view.

In comparison to all other approaches, we focus on the optimization of XQuery queries based on the schema of the input XML document in order to eliminate unnecessary query code, which computes not used intermediate results.

6 Summary and Conclusions

In this paper we have examined general optimization rules, which are applicable for all XQuery queries, and which are very useful for example in the scenario of XQuery queries on XQuery views. First, we have introduced a tester, which checks whether or not a given schema allows only XML documents, where the evaluation of a given XPath expression returns an empty set for all valid XML documents. In the second step, we have extended the tester. The extended tester optimizes variable assignments of XQuery queries so that computations of unnecessary intermediate results are eliminated.

We have shown by experimental results that the evaluation of the optimized queries saves processing costs depending on the amount of saved unnecessary intermediate results.

Future work will include further optimization rules, which, for example, also optimize the order of operations within the XQuery expression.

References

1. Altinel, M., Franklin, M. J.: Efficient Filtering of XML documents for Selective Dissemination of Information. In Proceedings of 26th International Conference on Very Large Databases, Cairo, Egypt (2000)
2. Chen, Y. B., Ling, T. W., Lee, M. L.: Designing Valid XML Views, ER 2002, LNCS 2503 (2002) 463-477
3. Deutsch, A., Tannen, V.: Reformulation of XML Queries and Constraints, In ICDT 2003, LNCS 2572 (2003) 225-241
4. Fan, W., Libkin, L.: On XML Integrity Constraints in the Presence of DTDs, Journal of the ACM, Vol. 49, No. 3 (2002) 368-406
5. Fernández, M., Kadiyska, Y., Suciu, D.: SilkRoute: A Framework for Publishing Relational Data in XML, ACM Transactions on Database Systems, Vol. 27, No. 4 (2002) 438-493
6. Fernández, M., Siméon, J., Chen. C., Choi, B., Dinoff, R., Gapeyev, V., Marian, A., Michiels, P., Onose, N., Petkanics, D., Radhakrishnan, M., Re, C., Resnick, L., Sur, G., Vyas, A., Wadler, P.: Galax pre-release 0.4.0 (2004) http://www.galaxquery.org

7. Fernández, M., Suciu, D.: Optimizing Regular Path Expressions Using Graph Schemas, Proceedings of the Fourteenth International Conference on Data Engineering (ICDE), Orlando, Florida, USA (1998)
8. Fisher, D., Lam, F., Wong, R. K.: Algebraic Transformation and Optimization for XQuery, APWeb 2004, LNCS 3007 (2004) 201-210
9. Gottlob, G., Koch, C., Pichler, R.: The Complexity of XPath Query Evaluation, In Proceedings of the 22th ACM SIGMOD-SIGACT-SIGART symposium of Principles of database systems (PODS 2003), San Diego, California, USA (2003)
10. Grinev, M., Kuznetsov, S.: Towards an Exhaustive Set of Rewriting Rules for XQuery Optimisation: BizQuery Experience, ADBIS 2002, LNCS 2435 (2002) 340-345
11. Groppe, S., Böttcher, S.: XPath Query Transformation based on XSLT stylesheets, Fifth International Workshop on Web Information and Data Management (WIDM'03), New Orleans, Louisiana, USA (2003)
12. Groppe, S., Böttcher, S., Birkenheuer, G.: Efficient Querying of transformed XML documents, 6th International Conference of Enterprise Information Systems (ICEIS 2004), Porto, Portugal (2004)
13. Groppe, S., Böttcher, S., Heckel, R., Birkenheuer, G.: Using XSLT Stylesheets to Transform XPath Queries. Eighth East-European Conference on Advances in Databases and Information Systems (ADBIS 2004), Budapest, Hungary (2004)
14. Hidders, J.: Satisfiability of XPath Expressions, DBPL 2003, LNCS 2921 (2004) 21 – 36
15. Kay, M.H.: Saxon - The XSLT and XQuery Processor (2004) http://saxon.sourceforge.net
16. Lakshmanan, L., Ramesh, G., Wang, H., Zhao, Z.: On Testing Satisfiability of Tree Pattern Queries, In Proceedings of the 30th VLDB Conference (VLDB 2004), Toronto, Canada (2004)
17. Marian, A., Siméon, J.: Projecting XML Documents. In Proceedings of the 29th VLDB Conference, Berlin, Germany (2003)
18. Oracle, Oracle XQuery Technology – Preview (2004) http://www.oracle.com/technology/tech/xml/xquery/index.html
19. Papakonstantinou, Y., Vianu, V.: DTD Inference for Views of XML Data, In Proceedings of the Nineteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2000), Dallas, Texas, USA (2000)
20. Paparizos, S., Wu, Y., Lakshmanan, L. V. S., Jagadish, H.V.: Tree Logical Classes for Efficient Evaluation of XQuery, SIGMOD 2004, Paris, France (2004)
21. Shanmugasundaram, J., Kiernan, J., Shekita, E., Fan, C., Funderburk, J.: Querying XML Views of Relational Data, In Proceedings of the 27th VLDB Conference, Roma, Italy, (2001)
22. Software AG, Tamino XML Server (2004) http://www.softwareag.com/tamino/News/tamino_41.htm
23. Wang, L., Mulchandani, M., Rundensteiner, E.A.: Updating XQuery Views Published over Relational Data: A Round-Trip Case Study, Xsym 2003, LNCS 2824 (2003) 223-237
24. W3C, XML Path Language (XPath) Version 1.0 (1999) http://www.w3.org/TR/xpath
25. W3C, XQuery 1.0: An XML Query Language, W3C Working Draft (2003) http://www.w3.org/TR/xquery